Decentralized Offload-based Execution on Memory-centric Compute Cores

Saambhavi Baskaran
The Pennsylvania State University
sxv49@psu.edu

Jack Sampson
The Pennsylvania State University
jms1257@psu.edu

ABSTRACT

With the end of Dennard scaling, power constraints have led to increasing compute specialization in the form of differently specialized accelerators integrated at various levels of the general-purpose system hierarchy. The result is that the most common general-purpose computing platform is now a heterogeneous mix of architectures even within a single die. Consequently, mapping application code regions into available execution engines has become a challenge due to different interfaces and increased software complexity. At the same time, the energy costs of data movement have become increasingly dominant relative to computation energy. This has inspired a move towards data-centric systems, where computation is brought to data, in contrast to traditional processing-centric models. However, enabling compute nearer memory entails its own challenges, including the interactions between distance-specialization and compute-specialization. The granularity of any offload to near(er) memory logic would impact the potential data transmission reduction, as smaller offloads will not be able to amortize the transmission costs of invocation and data return, while very large offloads can only be mapped onto logic that can support all of the necessary operations within kernel-scale codes, which exacerbates both area and power constraints.

For better energy efficiency, each set of related operations should be mapped onto the execution engine that, among those capable of running the set of operations, best balances the data movement and the degree of compute specialization of that engine for this code. Further, this offload should proceed in a decentralized way that keeps *both the data **and** control movement low* for all transitions among engines and transmissions of operands and results. To enable such a decentralized offload model, we propose an architecture interface that enables a common offload model for accelerators across the memory hierarchy and a tool chain to automatically identify (in a distance-aware fashion) and map profitable code regions on specialized execution engines. We evaluate the proposed architecture for a wide range of workloads and show energy reduction compared to an energy-efficient in-order core. We also demonstrate better area efficiency compared to kernel-scale offloads.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422778

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; **Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Reconfigurable computing**.

KEYWORDS

Memory-centric compiler optimization, data movement

ACM Reference Format:
Saambhavi Baskaran and Jack Sampson. 2020. Decentralized Offload-based Execution on Memory-centric Compute Cores. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3422575.3422778

1 INTRODUCTION

With the end of Dennard scaling, the slow-down in Moore's law, and the exacerbation of existing memory wall constraints in an era of data-intensive computing, further improvements in many computing platforms are increasingly becoming efficiency-bound [33]. Compute specialization is a traditional approach for increasing computational efficiency [19, 37, 77]. However, given the continued unequal scaling of logic and interconnect over the past several process generations, focusing on compute specialization alone is unlikely to achieve transformative efficiency changes absent commensurate specialization of data movement. Both the energy overhead of data movement and memory wall have necessitated systems to be increasingly data-centric, where the primary goal has shifted to customizing compute at different locations in the memory hierarchy (*distance specialization*) - including cache [2, 30, 52], memory [35, 69, 71, 80], and storage [26, 27, 68]. Thus, the design of future energy-efficient large-scale processing platforms must fundamentally involve a co-design of computational specialization and near(er) data computing techniques that balances the inherent tension between moving computation to a processing engine better capable of efficiently performing the upcoming operations and moving execution and/or data to where the upcoming operators are or will need to be, *because these two locations are not always the same.*

The addition of many specialized accelerators to the system, including integrated GPUs and FPGAs, has increased the host overhead in managing the control dependencies between various calls to different accelerators, and the communication between various compute units in the form of data transfers [6]. From a performance standpoint, it is not efficient for a high-performing IPC-optimized general-purpose core to just coordinate multiple offload calls. In view of this, existing specialization approaches assuming a host

issuing offloads onto different execution engines is inefficient. Considering that the cost of communication increases with distance between the host and co-processor units, although it can be balanced with increased offload task size, the primary reason for increased cost of offload is because the host is the central point of control throughout compute-specialized and in many distance-specialized models. This work addresses this problem by decentralizing acceleratable code regions such that the specialized hardware can sequence subsequent operations by itself rather than being entirely managed by the host. We exploit the well-known technique of chaining operators [23, 72], while ensuring distance-aware specialization of code regions. Enabling decentralized execution of accelerators entails the following primary challenges:

1. Determining candidate offloads: Given an arbitrary source code and an underlying architecture that does not define the function or kernel boundary, a challenge arises in identifying the best candidate to offload for an optimization metric.

2. Host-accelerator offload interface: The architecture interface should enable seamless low-cost operand and control transfer, when necessary. With more and more heterogeneous accelerators getting added at various points in the architecture, it is important that the interface remain inter-operable across architectures to improve programmability, and be future-proof.

3. Co-optimizing compute and distance: Identifying the code regions for specialization based on the location at which these need to be mapped to keep both control and data movement low.

This paper proposes a generic architecture interface and an automated profile-based compiler method for general purpose workloads with energy reduction from reduced control and data movement as the primary goal for a memory-centric heterogeneous architecture. The architecture interface enables a common asynchronous offload model that is independent of the offload granularity and distance of the compute engine from the host. The compiler identifies fine-grained code regions from arbitrary source code that are profitable in terms of reduced data and control movement, and generates runtime library variants based on the target execution engine. While the different execution platforms, namely general-purpose processors (GPPs), field-programmable gate arrays (FPGAs), coarse-grain reconfigurable arrays (CGRAs) or application-specific integrated circuits (ASICs) dictate the generality, area and energy efficiency tradeoffs of the accelerator architecture, our focus is more on identifying the offload candidates, and on determining how it should be mapped and invoked for better energy efficiency. For evaluation, we assume a system model that employs a CGRA fabric near-cache and near-memory. Our key contributions include:

- We posit a design space generated by compute and distance specialization axes and identify the design tradeoffs for future energy-efficient heterogeneous computing architectures.
- We present a generic architecture interface that enables energy-efficient management of control and data dependencies between multiple offloads to memory-centric computation cores.
- We propose a compiler framework that enables automatic identification, heterogeneous mapping, and co-placement of compute offloads onto execution engines with energy as the primary metric.
- We build an evaluation system with an in-order processor and CGRAs integrated near cache and memory, both of which support execution of heterogeneous offloads and compare the energy efficiency and performance of centralized and decentralized offload models for applications with varied locality characteristics. The results show that our proposed decentralized model can provide energy efficiencies and EDP improvements of $1.2\times - 3.9\times$ and $1.37\times - 4.4\times$ for cache-sensitive applications, and $1.17\times - 2.55\times$ and $1.4\times - 9.82\times$ for applications with streaming/random memory accesses, respectively.

2 BACKGROUND AND MOTIVATION

In this section, we broadly classify the evolved heterogeneous von Neumann architecture design space into three classes to analyze their qualitative trade-offs. We then identify the critical design factors and challenges for future computing systems, followed by presenting motivating experimental data for our approach.

2.1 Design Space and Taxonomy

There have been significant architecture innovations in the traditional von Neumann computational model evolving towards a heterogeneous model with processing elements of varying functions, granularities and architectures that are positioned in and near memory elements [7, 19, 29, 39, 67, 70, 77]. Figure 1a shows how we classify these approaches into three regions within the design space of combining compute and distance specializations. The horizontal axis describes increasingly narrow architecture specialization from left to right and the vertical axis represents the distinct points in the system hierarchy where compute can be mapped for a progressively better memory-centric system from top to bottom.

Classes 1a and b include architectures which primarily distinguish their specialization along the distance axis, retaining general or nearly general computation capability at all loci of executions. General-purpose processors exercise sequential or multiple flows of control mechanism as in multi-core or manycore processors to extract the parallelism available in the workloads. Decades of data specialization with low-latency caches and multiple flows of control kept the performance growing in the past era. The need to be data-centric has moved part of the control flow (single or multiple instruction offload [57, 79]) to be mapped onto near cache/memory (class 1a) or in-memory (class 1b) processors.

Class 2 includes architectures that exploit compute specialization for better performance or energy. The benefit from compute specialization can arise from the inherent efficiency in customizing the functional operator [41] and from exploiting the parallelism available in the application and/or from localizing the control flow within the target code region [59]. Code regions with high parallelism or with critical datapath operations are offloaded to customized hardware units, either for better performance [8, 14, 41, 75] or better energy [77]. Automatic (static [77] or dynamic [19]) or semi-automatic (user directives [45]) compiler mechanisms exist to allow exploitation of the underlying accelerator resources.

Classes 3a and b include architectures which specialize for computing near where the data is. The goal of class 3 architectures is primarily either to achieve better performance/energy efficiency by reducing the data movement. Processing in/near main memory has been shown to be beneficial both in reducing delay and energy for graph applications with low temporal reuse [4, 56]. On the other

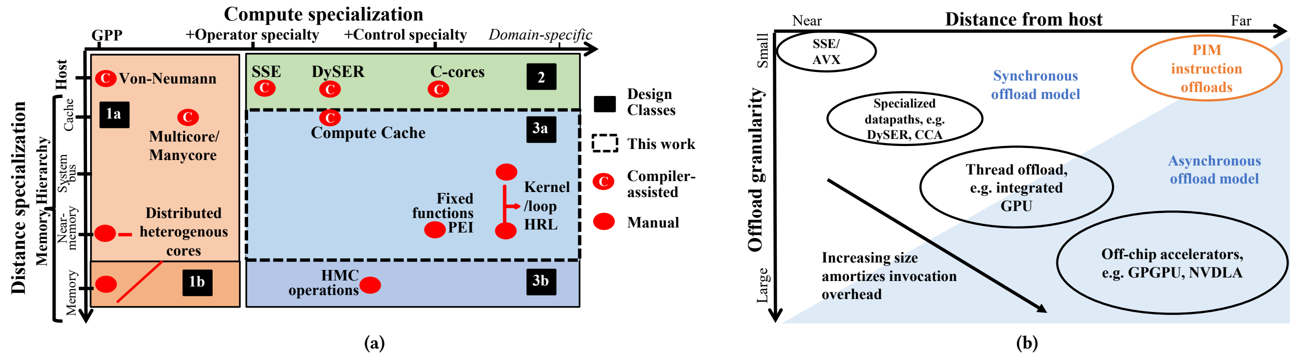


Figure 1: (a) Co-design space of compute specialization and distance specialization and (b) Relation between offload granularity and distance from the host. Design decisions in the taxonomy (a) have traditionally led to distance-proportionate granularity offloads seen in (b) to handle both data and control movement costs. As a result, the design choices are not independent of offload models required to move computation through various memory-centric processing engines.

hand, for applications with streaming or highly parallel memory access characteristics, manually identified kernels [7, 31, 34] or domain-specific loop offloads [43] have been shown to improve performance.

Summary: The general-purpose nature of class 1a and 1b architectures makes the computation itself energy-inefficient because it does not exercise compute specialization, and furthermore the energy and area requirements for achieving high performance (i.e. an IPC-optimized general-purpose core) are significant. Although the class 2 targets architectures providing better compute energy efficiency and performance, these approaches do not inherently specialize for distance. Sharing of data among physically distant specialized cores can increase data movement costs relative to a general-purpose solution, diluting or overshadowing the energy efficiency gains of datapath specialization. Although memory latency can be hidden in many cases by overlapping instruction executions, the data movement through the cache/memory hierarchy still exists irrespective of application reuse characteristics. While class 3 architectures enable moving along both axes of specialization at the same time, there are key design challenges in obtaining its potential benefits, which we discuss below. [1]

Automation: Current challenge of identifying candidate offloads is exacerbated by the increase in distance from the host. While operator specialization for potential execution can be determined statically, data usage patterns and control dependencies may be dependent on dynamic inputs and improper compute partitioning would lead to unnecessary data and control transfers between different loci of computations. Figure 1b displays the conventional relation between the offload model and the distance of the specialized compute engine from the host [15]. For the asynchronous accelerators in the lower right of the design space, offloadable code regions are usually identified manually [4, 7, 31, 34], as opposed to the synchronous accelerator space (upper triangle) which comprises of small granularity accelerators closely coupled with the host. Finding large granularity code regions within arbitrary applications is not widely automated in practice because of increased software complexity and scalability involved in hardware mapping. Loop accelerators are suitable only for perfect loop nests that are rare in arbitrary codes which have unknown loop bounds and complex control flows of runtime checks for vectorized operations.

Interfacing: While accelerators closely integrated with the host or with fixed function definitions profit from synchronous calls through low latency ISA extensions, with increasing distance from the host and with standardization, asynchronous accelerators use high-level libraries. Although customized ISA extensions reduce the control overhead, it would need a larger number of such extensions for coverage. Further, both function-customized ISA extensions and high-level libraries have inflexible function boundaries restricting the composability of multiple offloads [15].

Granularity: Technologies such as computing within SRAM and DRAM arrays have enabled bit-parallel computations [2, 30] and simple bounded operations [4, 35, 63]. Although being able to compute within memory structures can reduce application data movement, the granularity of such offloads is at odds with traditional offload models, as increased fine-grained offloads will escalate the dynamic control overhead for the host. Current proposals balance the invocation and communication latencies by virtue of application characteristics like graph applications (low temporal reuse) [4, 40] or by offloading larger granularity computations to amortize the latency [31, 34] as seen in Figure 1b. For a near-memory logic layer, function-level accelerator definitions can cause load balancing issues, and would need larger area while also demanding better implementation coverage of underlying accelerator architectures. Also, the area constraints make it harder to exploit the available bandwidth benefits with large granularity kernel-level parallelism.

2.2 The case for decentralization

In view of the fact that class 3 architectures specializing along both compute and distance are more energy-efficient, being able to decentralize compiler-identifiable offloads is essential to decouple the need to offload larger granularity and the underlying problem

[1]While our focus is on class 3a in this paper, we believe that the insights gained generalize for class 3b as well.

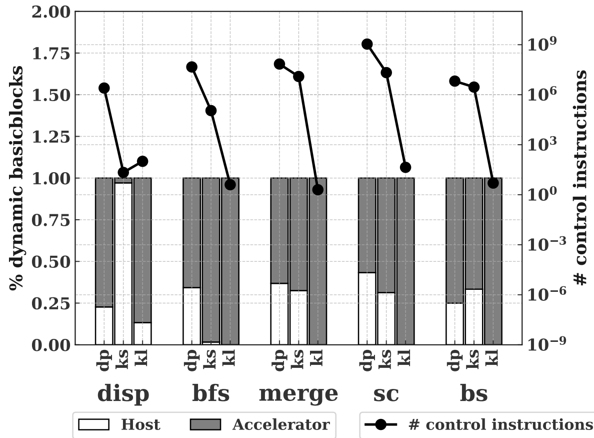


Figure 2: Breakdown of dynamic basic blocks and #control instructions for different granularities of host-controlled offload (*dp-datapath, ks-fixed function kernels, kl-large domain-specific kernels. See Section 4 for benchmark details)*

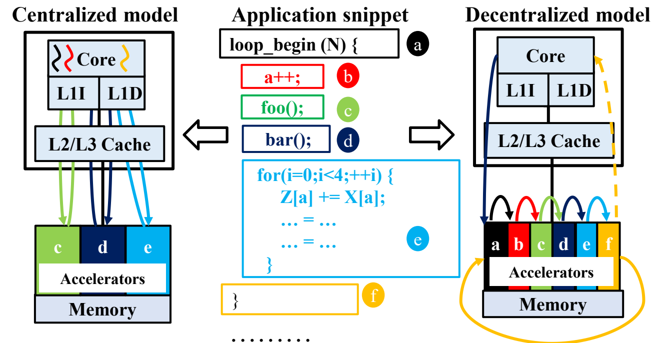


Figure 3: Centralized vs decentralized model

of distance-proportional invocation/communication overheads. *The goal is to achieve low control overhead irrespective of the size of the offload.*

Quantifying the control overhead: Figure 2 shows the division in the percentage of dynamic basic blocks that are executed by the host and near-memory accelerator along with the host control overhead (in number of instructions) for each offload configuration: ***dp***-compiler identified compound vector datapaths, ***ks***-manually identified general-purpose fixed function small granularity kernels and ***kl***-manually identified domain-specific large granularity kernels. We pick these configurations because, ***dp*** configurations are widely explored in the literature [19, 37, 46, 70] and are efficient at exploiting the ILP present; ***ks*** configurations are widely used as fixed function units near/in memory [4, 62]; ***kl*** configurations are large-granularity accelerator functions commonly deployed near system-bus or memory [20, 31, 34] [2]. Increasing granularity of offload raises the degree of operator specialization for most of the benchmarks except *disp* and *bs*. For ***dp***, a significant number of host control instructions are necessary for sequencing although most of the datapath operations are specialized. Although ***ks*** offloads are ideal to be implemented near memory, as seen in *disp*, it does not exploit all the opportunities to specialize for all applications. On the other hand, ***kl*** offloads incur very low control overhead due to specializing significant percentage of dynamic code regions involving both data paths and control paths. While manual identification of kernels can be useful, it is not scalable. From a performance perspective, multiple control instructions need to be issued in parallel for fine grained offloads such as ***dp*** and ***ks*** to reduce increased latency overheads owing to being distant from the host. Although ***dp*** specializes most of the datapath operations, it still incurs a lot of control overhead instructions for sequencing. *Note that the granularity specialization is often restricted with respect to distance. To be able to offload fine-grained operations or datapath, the architecture should allow low-overhead mechanisms.* We show that this can be reduced by being distance-aware in the process of identifying and mapping offloads.

Given that future architectures are going to have heterogeneous computing resources, it is important that the host neither become a performance bottleneck due to over-centralization in exercising control over all the other execution engines nor a source of energy overhead due to excessive data transmission for control to and from a central physical location. To maximize the computational utilization and increase energy efficiency of each of the variously specialized elements, sequencing of offloads to distributed compute elements also need to be specialized. Therefore, in the scope of this work, we focus on a platform with heterogeneous computing resources that can be distributed throughout the memory hierarchy, and aim to build an architecture and compiler framework to support a scalable decentralized execution model which is energy-efficient in both data and control movement.

3 ARCHITECTURE FRAMEWORK

This section provides an overview of the proposed decentralized, near-data heterogeneous architecture model. It describes the host-accelerator ISA interface followed by a description of how our evaluation architecture is implemented. It also outlines the compiler framework and execution model.

3.1 Towards decentralized architectures

Figure 3 shows an example of an application snippet with six parts - outer loop boundaries (ⓐ and ⓕ), increment operation (ⓑ), two accelerator kernels (ⓒ and ⓓ) and an inner *for* loop (ⓔ) along with plausible executions following both the centralized and decentralized execution models. In the traditional centralized model, assuming that the kernels and the inner loop are of sufficiently large granularities, both the kernels and loop will be identified as offloadable computations and the host will invoke these accelerators by transferring the needed arguments for the offload and invoking each of the accelerators individually. Although the three offloads are to be executed at the same location in the memory hierarchy, in the centralized model, the host must transfer both the data and control flow for every offload, which incurs energy and latency overheads. In the decentralized model the host needs to

[2]For ***ks***, an ISA extension for each fixed specialized function is assumed with limited arguments transferred as a burst to keep the invocation overhead low, as opposed to ***dp*** and ***kl*** implemented with a generic interface as described in Section 3.2.

initiate only the first offload and rest of the offload invocations are chained from the previous invocations. We list the fundamental characteristics of a decentralized model we aim for below:
(1) Lower invocation overhead: By being able to decentralize multiple offloads independent of the host, the effective work done by a group of accelerators is larger, which amortizes the invocation overhead for accelerators over large distances. Current near-memory acceleration approaches such as PEI [4], HRL [34] and NDA [31] target only fixed function blocks/kernels which limit composability. In such cases, decentralized models will help lower control overhead.
(2) Better area efficiency: Larger area requirements translate to load balancing issues and increased accelerator context overheads. While an entire loop body or kernel can be offloaded to amortize control overheads, the resulting area requirement will be higher. This cost can be prohibitive when co-running multiple accelerated applications and does not lend itself to the reuse of code patterns across multiple code regions in the same or different processes. With a decentralized model, not all parts of the offload need to be simultaneously active, thereby decreasing area requirements and adding dynamic flexibility in resource load-balancing among competing processes. Further, lower context overhead helps in migrating to different locations in the memory hierarchy dynamically.
(3) Heterogeneous fine-grained specialization: We abstract the control flows and datapaths independently, allowing both to be specialized differently along compute and distance axes. While the data paths can be mapped on to an ILP-exploiting dataflow accelerator or other fixed function hard blocks in the memory element, control heavy and low-ILP paths can be mapped onto processing elements that more heavily emphasize energy reduction than high performance in order to best achieve global energy-efficiency.
(4) Increased opportunities for specialization: If there was a non-acceleratable part in the loop in Figure 3, a centralized offload model would prevent offload of entire loop body, whereas a decentralized model could be made to return to a general-purpose processor only for that region.

We propose a compiler-hardware framework to accomplish decentralized fine-grained offloading with the compiler distinguishing between different data and control paths in the application, and the architecture enabling these offloads to happen in a host-transparent way irrespective of the size of each offload. In the scope of this paper, the compiler extracts all the accelerator functions as a library (detailed in Section 3.3). The compiler will perform heuristic-based offload accelerator extraction, emit a library of potential offloadable memory-centric accelerator definitions, and insert necessary control operations to activate offload sites when the necessary resources are available. Depending on where the offload must happen, the compiler may insert multiple shims into the original flow to allow for dynamic adaptation of the granularity of offload at runtime.

3.2 Architecture Interface

Figure 4 provides an overview of the key design components in supporting offload. The interface is intentionally generic to support a broad array of potential accelerator types of different granularities. A level of virtualization is provided so that applications can communicate with logical IDs for accelerators rather than physical IDs. While this virtualization imposes some additional requirements for OS involvement, it also simplifies using user-mode instructions for ISA-extensions for the accelerator communication interfaces, as well as simplifying revocation, interruption, and context switching at inter-accelerator invocation boundaries.

3.2.1 ***Interface Definition.*** The needs of the architecture interface to support decentralization are two-fold. Firstly, the interface should be able to support a wide granularity of accelerators ranging from simple vector operations to entire kernels as this ensures that any sequence of accelerators required to cover a given code can be invoked without requiring interface complexity to scale with the degree of heterogeneity. Secondly, where possible, the accelerator interface instructions should act asynchronously to support accelerators that are physically distant from the host without exposing the host pipeline to potentially large round-trip latencies. The first requirement allows the accelerator functions to be arbitrarily defined based on application needs and the second allows each accelerator to iteratively call itself [3] or another accelerator.

Although the interface can support a wide variety of processing engines spanning GPPs, reconfigurable or programmable fabrics and domain-specific fixed functions, as with many other accelerator interfaces, it requires a memory-mapped I/O interface (MMIO) sharing the host address space to expose the communication registers visible at each accelerator. Three *co-processor (cp)* instructions are defined as part of the generic user interface:

(i) ***cpset*** *fnid, regid, val:* An asynchronous instruction that is used to configure the input register *regid* of an accelerator executing a function *fnid* with value *val*.

(ii) val = ***cprun*** *fnid:* A non-blocking instruction that is used to invoke the execution of the function *fnid*. Optionally, it can also acknowledge the status of execution by returning a value, *val* once it is complete.

(iii) ***cpload*** *fnid, regid, n:* A blocking instruction to load *n* bytes starting from output register *regid* for the function *fnid*.

The *fnid, regid, val* fields are 16, 8 and 64 bits, respectively. The *cpload* instruction supports burst-reads. The key parameter *fnid* identifies the target acceleratable functionality, which is defined by the compiler and runtime libraries, thereby allowing flexibility of accelerator definitions. To let multiple offloads happen simultaneously and thereby exploit parallelism, the compiler distinguishes each static offload within the context of a thread with a distinct ID. While the instruction generality might compromise on the amount of control data that is transferred per offload, these can be mitigated by being able to share common register settings across multiple accelerators and with compiler support as explained in Section 3.3.1.

3.2.2 ***Interface Implementation.*** An accelerator functionality model with each accelerator having *n* accelerator resources (*AR*), an AR controller, configuration memory, and MMIO register interface as seen in Figure 4a is assumed. Each AR is assumed to be a partitioned logical segment of the hardware which is either defined at design time (e.g. a fixed-function unit implemented as an

[3] while general recursion could be supported with modest extensions to this model its exploration is beyond the scope of this work and we only consider tail-recursive invocations to simplify resource management complexity

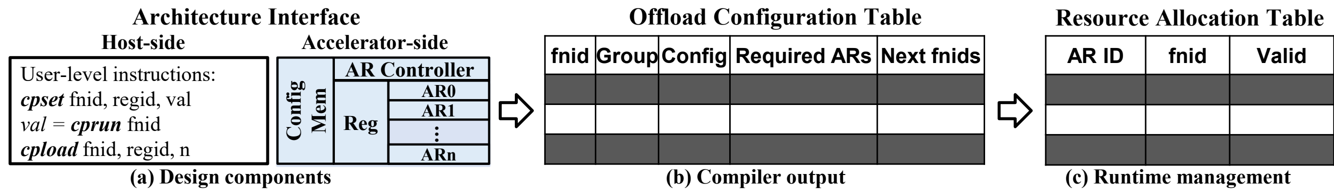


Figure 4: Key design components

ASIC in an SoC style heterogeneous design) or by the compiler (e.g. an NxN array of functional units in a CGRA or a reconfigurable partition in the case of an FPGA implementation). Note that, in most heterogeneous designs, not all accelerator definitions will be co-active for the currently running application, allowing for some space efficiency to be reclaimed via temporal multiplexing of resources. To be able to share register settings across multiple ARs, the register space is distributed and shared across a logical group of ARs. The grouping of accelerators ensures that multiple ARs can share the same registers and AR controller can maintain the ownership of the resources until freed by the application.

The compiler generates the *offload configuration table* (Figure 4b) as part of the application binary for the underlying hardware architecture. Each offload also has other attributes such as the logical definition of what constitutes an AR within the context of the application, required number of AR, group ID, and number of input and output registers. The interface definition does not limit the fields in the offload configuration table to enable the compiler and runtime to flexibly coordinate based on different aspects of the offloads.

The fundamental functionality of the AR controller is to manage the *resource allocation table* (Figure 4c) with valid mappings of each AR to *fnid*s and to load/configure the AR for the oncoming offloads at runtime based on the offload configurations table. The AR controller has access to a configuration memory which is loaded at program start time with the offload configuration table containing either the configuration bitstream and/or static register settings for all possible offloads for the application. The AR controller can raise an interrupt to the host if there were other exceptions with the executing contexts.

Chaining of Control and Data: To support decentralization of host control, the accelerator architecture allows the executing accelerator to chain-invoke another accelerator from the next possible *fnid*s, as defined in the offload configurations table. Empirically, the number of possible successor accelerators is limited, making chain invocation resource-allocation practical without substantial optimization in the applications we have studied. However, if the nodes in the graph of potential chaining destinations had particularly high arity, some form of resource prediction would likely be required to ensure high performance within practical resource budgets.

Accelerator Context: While, in the scope of this work, we assume that an accelerator can only invoke another upon completing its execution to maintain low context overhead, the interface neither requires the accelerator design to be preemptible, nor does it preclude preemption. In this work, we focus on a coarse reconfigurable fabric such as a CGRA for which the accelerator context constitutes the input and output register values. Since the register values are shared across accelerators, the register context must stay alive for the duration of the parent function/loop context. During a context switch, the OS coordinates with the AR controller to save the register values into the thread context.

3.2.3 ***Support for Accelerators in the Memory Hierarchy***.

While, conceptually, accelerators can be coherent or non-coherent with the host, to support near-memory execution, improve data locality, and avoid expensive cache traffic over off-chip pins, the compiler uses profiling data to identify memory regions accessed by potential offload sites and swaps the respective memory-allocation library calls with a custom implementation that allocates the data structures to a contiguous memory space [21, 50]. The *cpset* instruction is used to transfer the address translation for each distinct region to the address generation logic in the *AR controller*. During profile time, if the number of uncacheable CPU requests are more, meaning that there were significant non-offloadable code regions, the runtime cost model marks the offloads as non-profitable for near-memory execution. We extend the basic offload configuration table to also contain the coherency needs of the application and preferred location of execution. For near-memory execution with non-cacheable data, the CP instructions bypass on-chip accelerators to near-memory accelerators. Hence the host does not have to maintain or specify an explicit context of accelerator definitions and their locations.

3.2.4 ***Runtime***.

Multiple allocation policies can be used to acquire ARs for a given thread. When the host encounters a *cpset* instruction, the runtime software can acquire the minimum needed resources for that particular function from the OS and hand-over to the hardware AR controller, which can raise an exception to the host if additional ARs are needed for subsequent accelerator invocations. In this work, we assume that each thread is statically allocated with two ARs which are dynamically reconfigured by the AR controller during computation to hide the reconfiguration latencies.

3.3 Compiler Support

The compiler tool chain is shown in Figure 5a. In addition to the standard compilation steps, two more phases are added. At the front end, the application is optimized and converted to single static assignment (SSA) form. To maximize compute specialization, unrolling and vectorization are done besides the standard O3 optimizations. Function calls inside loops are inlined and *if* statements as part of feed-forward datapath regions are converted to predicated instructions where possible. In phase I, offloadable code blocks are identified based on cost-based analysis. In phase II, to decentralize the profitable offloads, the compiler identifies synthesizable code

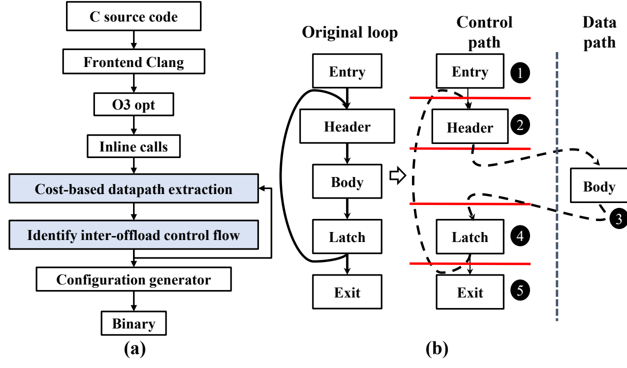


Figure 5: (a) Compiler tool chain (b) Control flow offloading

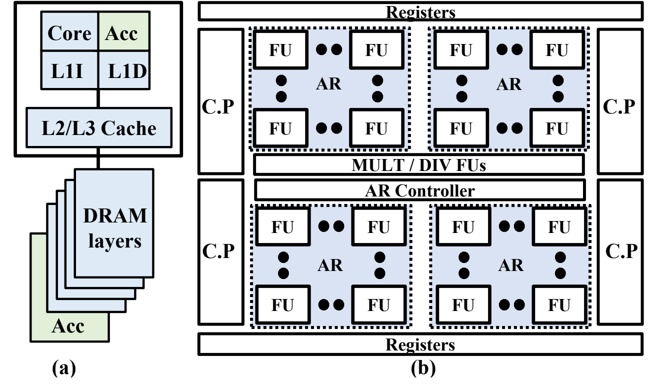


Figure 6: (a) Target architecture (b) Accelerator architecture

regions that, although they may be non-profitable in terms of parallelizable operations, exert control and data dependencies with the identified candidate offloads. In the backend, the offloaded blocks are mapped to the target accelerator architecture and the configurations are combined with the binary. We describe the two phases in detail below.

3.3.1 ***Feed-forward region selection (phase I)****.* In this phase, an offload is a set of feed-forward basic blocks (e.g. a hyperblock or path within a hyperblock) and the compiler looks for maximally exploiting the ILP available while also accounting for data movement. The cost of offloading is in terms of data movement like TOM [43], but unlike TOM, this phase of the compiler aims to offload feed-forward vector datapaths rather than highly parallel loops that are available in GPU workloads. An offload is profitable if the total number of bytes required to initialize the live-in register values is less than the amount of traffic to and from the nearby memory element. A runtime condition based on resource availability is added to either execute the software or offload version. Basic blocks are split around non-inlinable function calls to maximize the opportunities of finding offloads. A simple cost model in terms of data transfers and compute area cost is modeled, where a feed-forward region is outlined for possible offload if the communication cost is less than the cost of accessing the data across the memory hierarchy. Hence, at compile time, the condition: $i + x < M$, where, M is the number of data bytes accessed per execution of the region, is examined. Since there is a possibility that some of the live-in values will be constant across multiple invocations of the region, such values are hoisted so that the common values are of size i bytes. x denotes the number of bytes that vary across multiple invocations. While hoisting increases the data dependencies across the offloads in each context, it also reduces the host control overhead. For workloads with high temporal reuse and deep loop nest like matrix multiplication, the above equation will be insufficient. Hence, at profile time, the condition: $i + N * x < D$, is checked for profitability of offloads. The above equation ensures that the runtime initialization cost, N being the total execution count of the region, is less than the size of the data structures (D) used within the offload. Thirdly, the compute cost of specializing the set of blocks is calculated in terms of area and compared against the design constraint, if any. The synthesizability check assures that the region offloaded can be reasonably mapped onto configurable accelerator architectures and the host involvement is not necessary.

3.3.2 ***Control path outlining (phase II)****.* While outlining feed-forward paths enables the reduction of data movement, the repeated invocation and associated communication costs, depending on control flow characteristics, might offset the benefit of reducing data movement negatively, as these are fine-grained offloads. To reduce this cost, the compiler identifies control paths around the possible candidate feed-forward offloads and outlines these as individual offloads. Figure 5b shows a loop and how it gets divided into five fine-grained offloads. The flow of control is encoded as the logic outcomes of every offload, which is used during mapping stage or written to an output register to invoke the next accelerator. Identification of control flow is an iterative process with the number of iterations varying with different depth of loop nests and across multiple independent loops. For the evaluated workloads, a maximum of three levels provide considerable profits. To ease the runtime to schedule reconfiguration of all the blocks, the compiler creates distinct groups of the offloads within a code region (function or a loop nest) and adds a group ID as a hint to the runtime library. This also lets the accelerators within a region share the input and output register space.

3.4 Execution model

Figure 6a shows the target architecture for the study with accelerators near memory and near host. For our evaluation, we assume a CGRA fabric which is divided into groups of accelerator resources with multiple functional units (FU) as in Figure 6b. Each AR has an input and output queue (not shown in the figure) through which the memory operations pass. The steps of execution are set out below:

- The compiler-identified offloads are synthesized/mapped for the underlying accelerator architectures. The offload configuration table is loaded at program initialization time.
- When a *cpset* instruction is executed in a thread, the host forwards it to the nearest AR controller in the system hierarchy, where offload configuration table is looked up to identify the function to be configured. If the application uses uncacheable memory pages,

Table 1: Design configurations and benchmarks (*sim.run* denotes simulated #instructions in the region of interest)

Core	In-order, 1GHz
Cache line	64 Bytes
D cache	32 KB 4-way
L2 cache	256 KB 8-way, 8.2pJ per access
L3 cache	2 MB 16-way, 17.2pJ per access
On-chip crossbars	128-bit width, 2.4pJ/flit
Serial link	4 links of 10Gbps 2pJ/bit (8cm)
Memory	HMC 16-vaults, 4 layers, 32MB per vault, 2.47pJ/bit
Cache per vault	32 KB, 32nm, 20.9/24.7pJ per read/write access
Memory Crossbar	4-cycle latency, 256-bit flit [10], 1 GHz 28pJ/access
Accelerator	CGRA, 1 GHz

Benchmark	**Domain**	**Input dataset**	**Sim. run**
Disparity [76]	Stereo vision	288x352 images	9.84M
Hot**S**pot [17]	Structured grid	512x512 grid	9.35M
Feature **Trac**king [76]	Robot tracking	288x288 images	35.25M
Stream**C**luster [12]	Data mining	8192 points,64 point dimension	230.43M
Principle **C**omponent **A**nalysis [74]	Feature extraction	722x800 matrix	74.24M
Breadth **F**irst **S**earch	Graph analysis	scale-12,edge factor-32	2.15M
Robot **Loc**alization [76]	Image understanding	500 images of 88x5	88.79M
Scale **I**nvariant **F**eature **T**ransform [76]	Image analysis	288x352 images	213.56M
Merge sort [65]	Sort algorithm	512k elements of 8 bits each	67.14M
Kmeans [17]	Dense linear algebra	1000 objects; 36 attributes	27.33M
Singular **V**alue **D**ecomposition [74]	Feature extraction	250x250 image	241.74M
Latent **D**irichlet **A**llocation [74]	Natural language processing	251 docs; 12420 terms	153.51M
Black**S**choles [12]	Financial analysis	655KB	170.39M

the runtime with OS support allocates the number of ARs required near memory and forwards the *cpset* to the AR controller near memory which in turn configures the ARs. The runtime system routes the following *cpsets* and *cprun* in the given thread to the same location. If the offload does not use uncacheable memory page flag, then the offload is placed near host.

• Upon receiving a *cprun* from the AR controller, the accelerator begins execution. Compute, register, and memory operations mapped in parallel are executed simultaneously. Reads and writes to memory are routed as packets through the AR queues for enforcing memory ordering. The ARs support multi-cycle memory and compute operations through a valid-ready interface between multiple *FU*s. The AR controller performs reconfiguration of the subsequent accelerator functions, if any, based on the offload configuration table in parallel with the computations.

• The *cpload* instruction blocks until the accelerator finishes execution and it can be issued ahead to overlap the communication latency with the execution of the accelerator.

• As the accelerator finishes execution, the status is updated in its output register, based on which the AR controller invokes successive AR calls or issues an acknowledgement to the host or responds to the *cpload* instructions from host. Optionally, an accelerator chain-invokes another in a sequence if these are placed and mapped statically within one AR.

3.4.1 ***Discussion****.* Since we target generality over specificity, we show heterogeneity in the kinds of operations that get specialized rather than in the kind of substrate such as FPGA, CGRA, programmable or domain-specific hardware. While this may not strictly maximize the efficiency gains within a given accelerator pipeline, we expect the insights gained into the benefits of the offload model itself to generalize across different implementations of the accelerators themselves. In the context of this work, the loci of computations are specified to be *near-memory* or *near-L1 cache* based on offline profiling. While the offloads themselves are dynamically reconfigured and invoked during the execution, the hardware or runtime can further be extended to incorporate hardware structures that monitor the memory access characteristics and thereby dynamically decide the placement of offloads either based on application locality characteristics and/or resource availability.

In this work, our evaluation focuses on near-memory/cache, rather than multi-placement accelerator models. However, the compiler-driven approach and architectural interface presented are general and could also be applied to models that include offloads to multiple levels of the memory hierarchy [16, 52]. Additionally, while we have only discussed offloads within a single application, the AR controller and runtime system could be further enhanced to support dynamic resource allocation based on load balancing across multiple simultaneously executing applications, as the fundamental interfaces are already virtualized.

4 METHODOLOGY

To evaluate the generality of our proposed decentralized offload model, we choose applications from varied domains with different locality characteristics and multiple architecture configurations.
Simulation framework: The input C/C++ applications are compiled using a compiler framework based on the LLVM compiler infrastructure (9.0.0) [48]. Based on profiling runs, we identify function calls that need to be inlined to reduce the overheads of toggling between multiple loci of computations. The framework is extended with passes to identify and outline offloadable code regions for hardware mapping (as discussed in Section 3.3). We use the compiler's intermediate representation (IR) for hardware mapping. During the mapping stage, instructions in each basic block are topologically ordered and placed on a CGRA with an NxN array of functional units following a greedy approach.

For performance estimations, binary instrumentation is done to generate a dynamic instruction trace containing computation and memory events from both in-order core and accelerator executions. The compute timing annotations are relative to the memory events and are based on instruction-CGRA FU mapping and critical path analysis. The captured compute and memory events are replayed in Gem5 [13] respecting the original program's memory dependencies for the modeled memory system as per the approach by Nilakantan et al. in Synchrotrace [58]. Table 1 shows the parameters of the evaluated system. For configurations with accelerators near the cache hierarchy, the accelerator shares the L1 cache with the host. Memory is modeled as a hybrid memory cube with 16 vaults and 4 ranks per vault [9, 62]. The logic layer contains a 32 KB cache for each vault and a crossbar interconnect connecting the accelerators to the vault caches. We assume a 22nm technology for the host processor, caches, interconnects, and CGRA on-chip. We conservatively model the CGRA and the caches on the logic layer of the HMC with a 32nm technology.

Host core, cache hierarchy, and interconnect energy are estimated using McPAT [49]. Cacti 7 [11, 18] is used to estimate the energy for near-memory cache, configuration cache, memory access queues in the accelerator, and 3D memory. Serial energy is assumed to be 2 pJ/b following previous literature [1, 31, 62]. The CGRA functional unit energies are scaled [73] from previous literature [31] for a LP 0.9V 32nm technology.

Benchmarks: We target C/C++ applications and kernels from Cortex/SDVBS [74, 76], Parsec [12], Machsuite [65] and Rodinia [17]. These benchmarks have varied data sizes, and each is simulated for regions of interest, the size of which is shown in the Table 1 in terms of number of dynamic IR instructions in the O3-optimized software baseline. We select applications with limited dependencies on external libraries and complex data structures, as the automation aspects of our tool are in the early stages of development, and complex library dependencies do not currently synthesize to hardware efficiently.

System configurations: We consider the following architecture configurations:

- ***In-order host (H-SW)***: In-order processor without accelerators
- **Centralized datapath operator specialization**: Compound vector feed-forward datapaths mapped on CGRA fabric with the host controlling each offload decision:

→ ***Near-cache centralized (C-C)*** *(DySER-like [37])*: Accelerators near L1-cache with a centralized host having 2-cycle overhead of writing its register

→ ***Near-memory centralized (M-C)*** *(PEI-like [4])*: Accelerators near-memory with a centralized host having 20-cycle overhead of writing its register. The granularity of offloads in this configuration is like PEI for graph, sort and streamcluster benchmarks. While PEI implements the parallel version of BFS, which in turn helps in amortizing the access latencies, we use the sequential implementation and group multiple operations to amortize the initialization and invocation costs of this centralized fine-grained offload model.

- **Decentralized datapath + control flow specialization**: In addition to feed-forward compound vector datapaths, these accelerators support chaining of other accelerators (discussed in Section 3.2.2) upon their completion. We evaluate various degrees of decentralized offloads (as discussed in Section 3.3.2) with diverse area requirements:

→ ***Near-memory decentralized - degree-1 (M-D1)***: Accelerators near-memory with up to two iterations of control flow offloading.

→ ***Near-cache decentralized - degree-2 (C-D2)***: Accelerators near-L1 cache with three or more iterations of control flow offloading.

→ ***Near-memory decentralized - degree-2 (M-D2)***: Accelerators near-memory with three or more iterations of control flow offloading.

→ ***Area constrained near-memory decentralized - degree-2 (M-D2+AC)***: We assume an area-constrained static CGRA mapping on top of the previous configuration only for logic near memory.

All the system configurations with accelerator executions use our proposed architecture interface and compiler framework from Section 3.

Table 2: Geometric mean and maximum (in brackets) metrics normalized to H-SW configuration over all applications

	H-SW	C-C	C-D2	M-C	M-D1	M-D2	M-D2+AC
Energy efficiency	1.00 (1.00)	3.45 (9.10)	**5.39** (**24.22**)	1.97 (5.48)	3.39 (6.45)	3.48 (5.75)	3.50 (6.42)
Speedup	1.00 (1.00)	1.83 (2.90)	2.16 (4.04)	1.56 (7.16)	1.89 (**7.57**)	**2.32** (7.51)	1.90 (7.55)
EDP improvement	1.00 (1.00)	6.33 (26.42)	**11.64** (44.86)	3.06 (39.24)	6.41 (**48.82**)	8.06 (42.61)	6.66 (48.47)

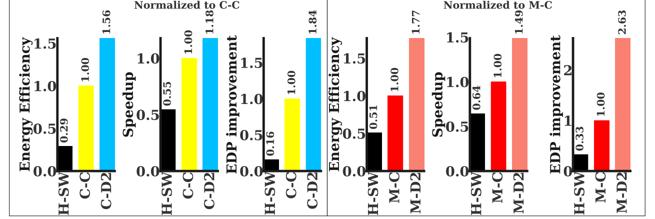


Figure 7: Geometric mean of normalized metrics for centralized versus decentralized configurations over all applications

5 RESULTS

We emphasize that our primary goal is to evaluate the efficiency of decentralized offload decisions and understand its potential and tradeoffs while specializing at near-cache and near-memory locations. This section illustrates the major benefits of decentralization and provides detailed analysis of the sources of performance improvements. We demonstrate that the overhead costs of decentralization in terms of area and reconfiguration energy are nominal.

5.1 Major Benefits from Decentralization

Table 2 compares the geometric mean and maximum (in brackets) achieved energy efficiencies, speedups and EDP improvements for all architecture configurations normalized to the baseline in-order configuration (*H-SW*) over all applications. The data shows that compute specialization of *datapath alone - C-C and M-C* can provide an energy efficiency of **3.45× and 1.97×** compared to an in-order core (*H-SW*), while the performance improves by **1.83× and 1.56×**, respectively. Specializing the control path further with decentralization degree of two improves the energy efficiency of near-cache *C-D2* and near-memory *M-D2* configurations by **5.39× and 3.48×**, while speedups improve by **2.16× and 2.32×**, respectively. For applications with cache resident datasets and high reuse distances, decentralizing near-cache can achieve a maximum energy efficiency of **24.22×**, while for applications with streaming or random memory accesses decentralizing near-memory can achieve a maximum speedup of **7.57×** along with an EDP improvement of **48.82×** compared to *H-SW* baseline.

While these results are from evaluating applications of varied data locality characteristics, the key points of comparison are between the centralized and decentralized configurations, namely C-C versus C-D2 and M-C versus M-D1/2, which show the potential of localizing control in a way that is amenable to compiler automation while also keeping energy and area constraints in perspective. Figure 7 shows that the host decentralized configuration *C-D2* has a

Table 3: Control movement and area statistics for M-D2+AC configuration and area of function-sized offloads. Columns: # **Dynamic Offloads** shows the # offloads from processor and from chaining; **Avg. run length** (operation cycles without memory latencies) gives the average critical path latency of a chain of offloads and of one offload; **#Registers** is the range of register space required for all offloads in the application; **Offload area** is the range of FUs required per offload definition; **Function area** gives the range of FUs required for a function if the entire function is to be offloaded

Apps	# Dynamic Offloads (Control movement)		Avg. run length (cycles)		#Registers (Context size)		Offload area (FUs)			Function area (FUs)		
	host	chain	chain	offload	min	max	min	max	avg	min	max	avg
disp	13	451943	874590	25	1	30	8	361	56	187	3286	769
hots	2	381312	7336715	38	12	38	14	237	100	1034	1034	1034
trac	164	926436	258109	45	1	37	8	445	61	232	6841	1268
sc	46503	217595595	298303	63	3	32	11	210	63	76	43986	5148
pca	4	1677629	22166684	52	2	21	13	312	57	686	2825	1601
bfs	2	1594654	12229324	15	3	15	19	45	30	116	157	136
loc	5394	8712107	47649	29	2	43	9	422	64	129	2857	668
sift	2757465	79806755	915	30	1	49	7	683	61	232	12042	1401
merg	3	14207028	78117699	16	3	84	14	329	78	202	1911	975
kmea	4	5279363	34324644	26	2	27	13	103	28	131	1258	527
svd	5	30844855	346689590	56	3	46	11	800	65	355	9529	2560
lda	249450	75626505	13745	45	0	27	6	202	55	11	3567	727
bs	1	4915500	571802602	116	1	7	13	184	74	209	209	209

speedup of **1.18**× and energy efficiency of **1.56**× compared to the centralized configuration-*C-C*, while near-memory decentralized configuration *M-D2* has better speedup and energy efficiency of **1.49**× and **1.77**× compared to *M-C*, respectively. *The decentralization of control helps in reducing the processor-accelerator communication overheads and the benefits of decentralization increases with increasing distance of the accelerator from the host.*

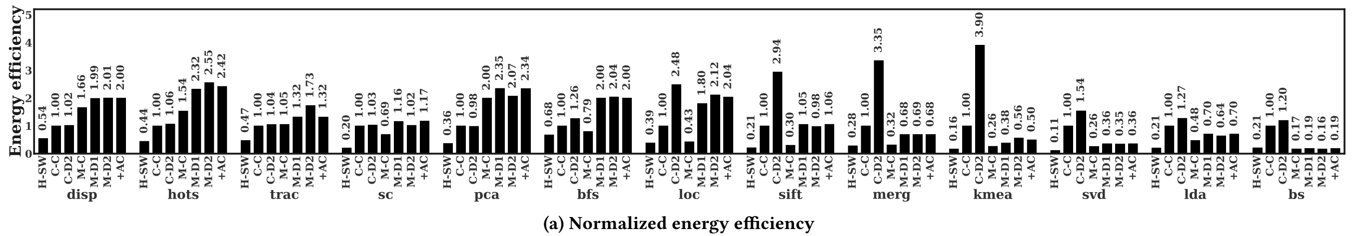
To illustrate what this means for near-memory accelerators, Table 3 quantifies the control movement and reduction of area requirements in terms of CGRA functional units (FU). In the context of this table, the number of dynamic offloads is considered as a metric of control data movement. For most of the applications, the host overhead of offloading forms only a small percentage of the chaining overhead (less than 1% for all and up to 4% for *sift*). Prior studies mitigate this offload overhead by increasing the size of the offload or by issuing multiple offloads in parallel. While larger sized offloads are difficult to extract and need higher area requirements, issuing multiple offloads still incurs the control movement overhead. Assuming a simple design without additional pipelining hardware, a kernel-sized offload would require a maximum of **43K units**, whereas *M-D2+AC* configuration requires only up to a maximum of **800 units**. The table also shows the average run length of a single offload to be in the range of **16-116** clock cycles, while the average run length of a single chain of offload is in the range of **887-571M** clock cycles. The number of registers required per accelerator (**0-84**) translates to the context size of the offload. *Overall, decentralization of offloads helps in reducing the control data movement while also maintaining fine-granularity offloads with reduced area requirements.*

5.2 Detailed Performance Analysis

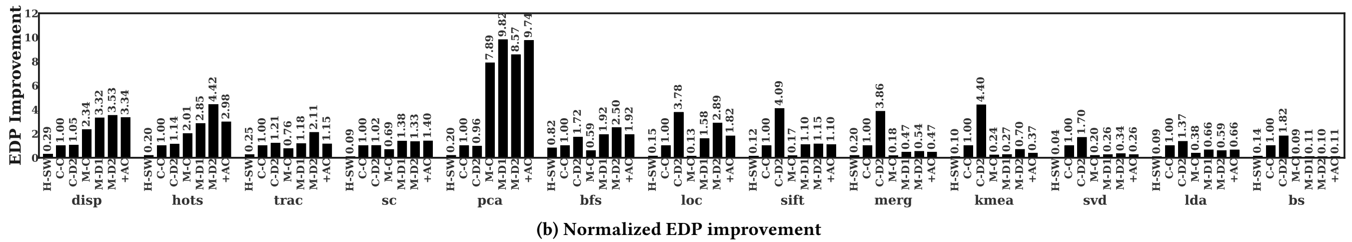
Since we have applications with diverse locality characteristics, we discuss the sources of performance variations below. Energy efficiency and EDP improvement are shown in Figures 8a and 8b, respectively. We present the results normalized to the near-cache datapath-only offloads *C-C* to contextualize the general efficacy of our decentralized offload decisions. In general, the *decentralized* configurations (*C-D2 and M-Dx*) show consistently better energy efficiency in the range of **1.17**×**-3.9**× and EDP improvement in the range of **1.37**×**-9.82**× than the *C-C* configuration.

There are two classes of applications we consider, since the near-memory caches are not configured to accommodate large reuse distances. Applications with low reuse distance or random memory accesses (*disp, hots, trac, sc, pca, bfs and loc* termed as Group-1) show better energy efficiency and EDP improvement in architectures with distance specialization near-memory, while the other applications with longer reuse distance and/or cache-resident workloads (termed as Group-2) perform better with near-cache specialization.

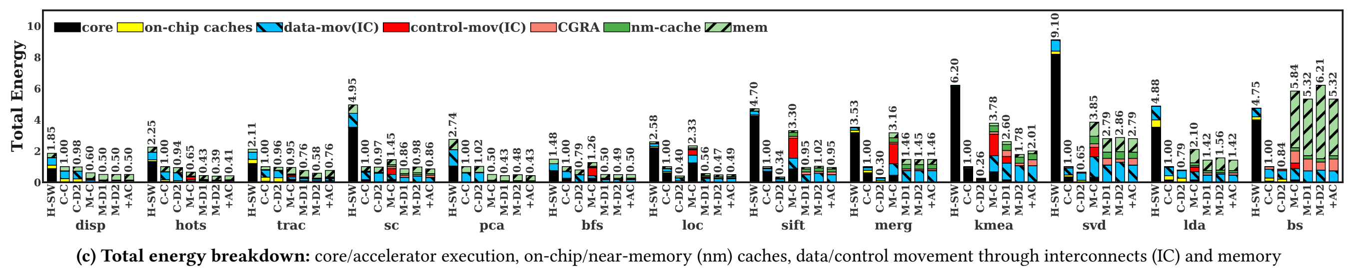
Effect of data and control movement: The energy efficiency of the *M-C* configuration for *disp, hots, trac, and pca* is due to both compute specialization and data localization, and is in the range of **1.05**×**-2.0**× compared to baseline *C-C*. The reduced energy consumption from data movement through serial link and on-chip interconnects can be seen in Figure 8e and Table 4. However, in the case of *sc, bfs, and loc*, the energy costs of control data movement through serial-link, (*control-move (SL)* component in Figure 8e), are higher than the benefit from localizing application data. *M-C* also has serial link energy overhead due to any non-cacheable data movement in non-offloadable code regions as seen in *loc* benchmark. Overlapping computation with control transfer, where applicable, could amortize the delay overhead for the *M-C* configuration, although the data/control movement will remain. In these cases,



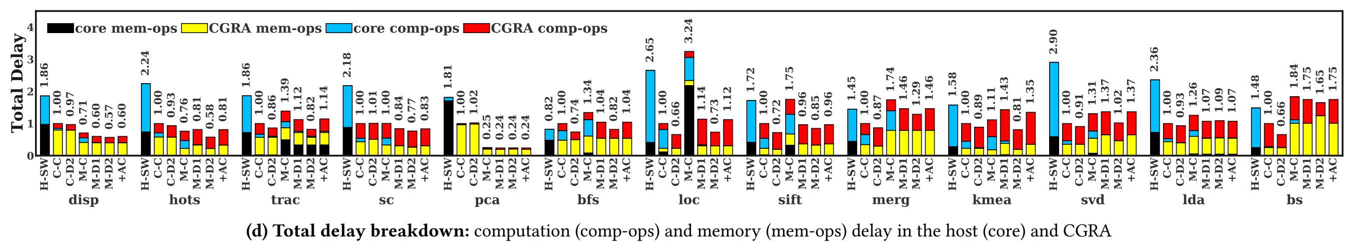
(a) **Normalized energy efficiency**



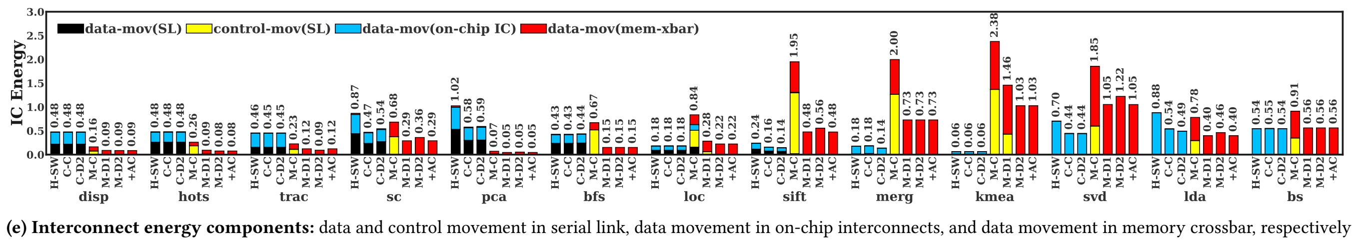
(b) **Normalized EDP improvement**



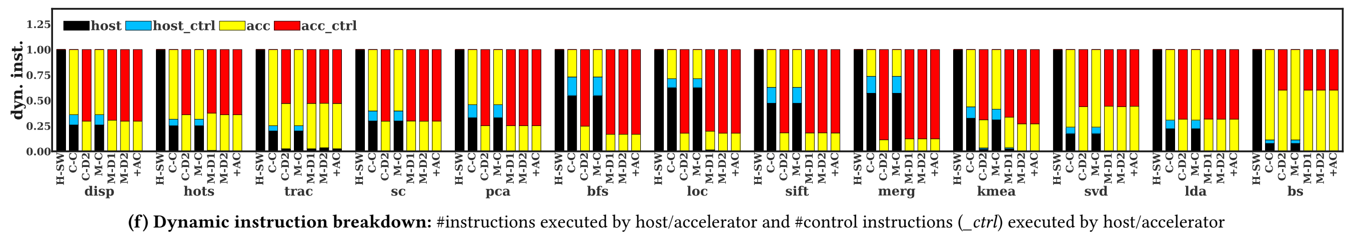
(c) **Total energy breakdown:** core/accelerator execution, on-chip/near-memory (nm) caches, data/control movement through interconnects (IC) and memory



(d) **Total delay breakdown:** computation (comp-ops) and memory (mem-ops) delay in the host (core) and CGRA



(e) **Interconnect energy components:** data and control movement in serial link, data movement in on-chip interconnects, and data movement in memory crossbar, respectively



(f) **Dynamic instruction breakdown:** #instructions executed by host/accelerator and #control instructions (_*ctrl*) executed by host/accelerator

Figure 8: Key evaluated metrics (with breakdown) for various architecture configurations and benchmarks

decentralized *M-Dx* configurations reduce the energy overhead of control transfer and hence have better energy efficiency and performance (*disp, hots, trac, sc, pca, bfs, loc, and sift*) compared to *M-C and C-C*. Decentralizing more than three iterations of control flows brings an energy reduction in *hots, trac, and loc* due to the decrease in data and control movement. Comparing *C-C versus M-Dx*, all applications show reduction in data movement energy through on-chip cache hierarchy and on-chip interconnect as seen in Figure 8e. However, for cache-sensitive applications, this translates to increased data movement through the near-memory crossbar (Figure 8e) and additional memory energy because the near-memory caches do not capture accesses with long reuse distance.

While *C-C* is nearer to host and has comparatively better control locality compared to *M-C*, it must fetch the data all the way through the cache hierarchy. As a result, the data movement through cache hierarchy and serial link still exists. The energy reduction in *C-D2* compared to its centralized configurations is attributable to reduced latency overhead of control transfer and communication between host and accelerator as seen in reduced latency in the host computation (*core comp-ops* component in Figure 8d).

M-D1 and M-D2 perform better than C-C for applications with random/streaming memory accesses, whereas C-D2 performs better than C-C for cache-sensitive applications. Generally, decentralization reduces energy overhead costs for near-memory offloads and reduces latency costs for near-host offloads.

Host Dynamic Instruction Reduction: Figure 8f shows the instructions broken down into arithmetic/logical compute or memory operations versus control instructions executed by host and accelerator. *Host and acc* refer to the dynamic compute/memory instructions executed by the host processor and accelerator, respectively. *Host_ctrl* accounts for the number of branch and offload instructions executed by the host. *acc_ctrl* accounts for the chained offload instructions of accelerators. Although the *C-C and M-C* configurations specialize for datapath operations, the host still executes a significant proportion of dynamic instructions. The decentralization of control in *C-D2, M-D1 and M-D2* reduces the dynamic instruction overhead of the host significantly (for instance, *disp* shows a ~25% reduction). The accelerator control overhead accounting for co-processor instructions gets converted into logical outcomes written to adjacent registers or forwarded to adjacent functional units when multiple accelerators are mapped within an AR. Hence, the execution of co-processor control within a local accelerator group are overlapped with computation and both the latency and energy overheads of these instructions are negligible for the workloads considered. *Decentralization increases the percentage of execution time expended on an accelerator to ~99%.*

Bandwidth Utilization: *Memory bandwidth:* Besides compute specialization and localized control/data, offloading datapath operations near memory (*M-C, M-Dx*) optimized for instruction-level parallelism shows both better energy efficiency and performance for group-1 applications owing to being able to exploit high bandwidth near memory, seen as reduced memory delay (*CGRA mem-ops*) in Figure 8d for *disp, hots, trac, sc and pca.* Due to contention in the crossbar and higher memory traffic, *sift, sc, lda and pca* show increase in energy for *M-D2* configuration, and *pca and lda* show reduced speedup for *M-D2* than *M-D1. Off-chip bandwidth:* Table 4 shows fine-grained data movement in the system hierarchy. The

Table 4: Data movement in the system hierarchy. Columns (3-8): #bytes read and written to L1 from host or near-cache accelerator, #bytes from L1 to L2, L2 to L3, L3-offchip serial links, #bytes from both serial links and accelerator to near-memory cache, and #bytes accessed in DRAM

App	Config	H/A-L1	L1-L2	L2-L3	L3-SL	SL/A-NM$	NM$-M
disp	C-C	34.53M	37.36M	44.19M	24.26M	-	24.24M
	M-C	9.64K	9.94K	9.94K	9.94K	34.52M	37.33M
	M-D2	260	392	392	392	34.53M	37.33M
hots	C-C	10.73M	11.72M	8.18M	7.61M	-	7.61M
	M-C	0	0	0	0	10.73M	8.91M
	M-D2	0	0	0	0	10.73M	8.91M
trac	C-C	66.93M	122.47M	75.53M	31.52M	-	31.49M
	M-C	3.06M	2.45M	2.45M	2.45M	63.87M	77.66M
	M-D2	1.83M	1.63M	1.63M	1.63M	53.85M	58.75M
sc	C-C	252.45M	138.80M	138.87M	123.29M	-	122.37M
	M-C	2.34M	263.94K	263.94K	263.94K	250.12M	138.22M
	M-D2	1.54M	1.82K	1.82K	1.82K	289.98M	158.23M
pca	C-C	368.39M	1403.78M	1403.75M	1403.73M	-	1401.96M
	M-C	92.12K	102.31K	89.83K	90.09K	368.30M	1400.83M
	M-D2	0	0	0	0	376.27M	1431.89M
bfs	C-C	5.02M	4.98M	4.71M	4.42M	-	4.41M
	M-C	116.24K	116.24K	116.24K	116.24K	4.90M	4.80M
	M-D2	34	34	34	34	5.02M	4.80M
loc	C-C	203.59M	44.09M	28.67M	26.70M	-	26.69M
	M-C	62.66M	52.00M	51.75M	50.73M	140.93M	33.76M
	M-D2	122.32K	156.04K	136.71K	136.71K	203.47M	33.70M
sift	C-C	202.61M	25.72M	27.90M	18.63M	-	18.62M
	M-C	5.69M	5.69M	5.69M	5.69M	196.92M	25.43M
	M-D2	216	272	272	272	176.40M	23.73M
merg	C-C	49.81M	46.58M	43.73M	1.05M	-	1.05M
	M-C	0	0	0	0	49.81M	44.44M
	M-D2	0	0	0	0	49.81M	44.44M
kmea	C-C	69.31M	4.96M	157.89K	158.21K	-	158.21K
	M-C	1.49M	10.58K	10.58K	10.78K	58.84M	4.48M
	M-D2	16	16	16	16	60.34M	4.72M
svd	C-C	362.68M	281.62M	382.66K	285.31K	-	285.31K
	M-C	1.39M	1.38M	1.38M	1.38M	361.29M	137.29M
	M-D2	8	8	8	8	354.37M	137.14M
lda	C-C	459.67M	598.15M	230.02M	1.98M	-	1.97M
	M-C	4.95M	3.16M	3.16M	3.17M	454.72M	315.98M
	M-D2	2.20M	2.15M	2.15M	2.15M	419.92M	288.67M
bs	C-C	216.27M	362.69M	52.39M	459.46K	-	459.46K
	M-C	12	128	128	128	216.27M	659.46M
	M-D2	12	128	128	128	216.27M	856.07M

energy consumption over the serial-link due to both control and data reduces for *M-D2* configuration in group-1 applications, which translates to off-chip bandwidth savings seen in the *L3-SL* component in the table. Comparing *M-C and M-D2* shows that, irrespective of the type of interconnect (2.5D or 3D) between the accelerator and memory chip, decentralization helps in localizing data movement to the computation, thereby offering both performance and energy advantages.

5.3 Reducing Overheads of Decentralization

Area: As shown in Table 3, the range of area overheads for the decentralized offloads within each application is much smaller than the area required if each function were to be placed entirely. To

reduce the area overheads further for logic layer implementation, instructions within an offload are mapped to a 40x40 AR fabric of functional units capable of performing integer/floating-point addition, subtraction, and logical operations. We assume two ARs for the area-constrained configuration *M-D2+AC* and the number of pipelined multiplier and divider units is limited to one for every two ARs. The area required for a 40x40 AR along with two multiplier/divider units, memory access queues and interconnect switches is $0.41mm^2$ per vault, and a near-vault cache takes $0.086mm^2$ for a 32nm technology. Additionally, the geometric mean of the static CGRA instruction size of all the considered applications is 7.2KB, and the area for a configuration cache of 16 KB takes $0.04mm^2$. Offloads with width higher than 40 are wrapped around and with depth higher than 40 are split across multiple ARs. The control path offloads are placed statically within AR based on a greedy policy. For longer control paths, the instructions are mapped onto a functional unit, denoted by *CP* in the Figure 6b, which executes the given instructions sequentially. Adding a resource constraint of only two ARs introduces a performance overhead of **1.2**× for *M-D2+AC* with respect to *M-D2* configuration. In cases where the design is constrained by near-memory crossbar or memory traffic, area constraining helps in reducing the energy as seen in Figure 8c for the *sc, sift, pca, svd, lda and bs* benchmarks.

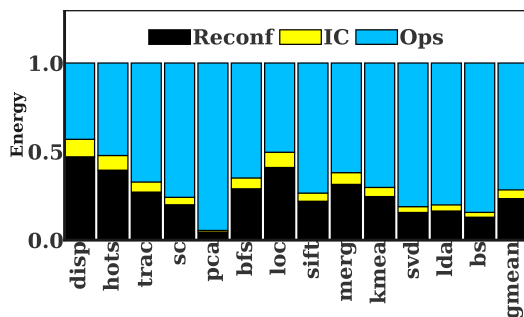


Figure 9: Dynamic energy breakdown of CGRA

Reconfiguration: With the configuration bits for intra-AR functional unit and switch being six and three bits, respectively, Figure 9 shows the CGRA energy components for functional units (*Ops*), intra and inter-AR interconnects (*IC*) and reconfiguration (*Reconf*). Reconfiguration and interconnect constitute 25% and 5.3% of the total CGRA energy (geometric mean), respectively. For the given applications, the oncoming AR reconfiguration is pipelined with the ongoing AR computation. For other applications with a high branching factor of accelerator invocations, in addition to resource prediction, statically placing the critical blocks first and pipelining reconfiguration at a finer granularity of the oncoming AR functions can reduce the reconfiguration latency overhead during chaining.

6 RELATED WORK

Our work spans three main areas: compute specialization, near-data computing, and accelerators with specialized control flows.

Compute specialization: The rise of single-application silicon markets has made specializing computations via diverse dedicated hardware accelerators economically viable [32, 44, 53, 60]. Catapult [64] seeks to accelerate datacenter applications like web search with reconfigurable architectures. ASIC clouds [53] are made of large arrays of ASIC accelerators designed to be TCO-optimal for datacenters. Despite improved performance and energy efficiency, increased specialization makes it difficult to use resources more efficiently outside the target area [47]. On the other hand, programmable specialization architectures cater to both generality and specialization. DySER [37] targets highly parallel code regions and dynamically specializes these to deliver significant performance benefits for a broader range of application domains. Unlike specializing parallel and regular code regions, conservation-cores [77] are designed to improve energy efficiency for hot irregular code regions. By the nature of these architectures, most of these accelerators are closely coupled with the processor for control sequencing and data sharing, and hence do not exercise distance specialization.

Near-data architectures: The trend towards data-centric computing has prompted research towards many near-memory specialized architectures [3, 7, 28]. HRL [34] and NDA [31] propose to map candidate offloads on a coarse reconfigurable unit (CGRA) near memory. Manual programmer changes are not a scalable solution. Further, while kernel offloads may be compute-intensive, they need not necessarily be memory-intensive as well. Secondly, not all parts of the kernel need to be concurrently executed, causing under-utilization of the area. IRAM [61] assumes a co-processor tailored to support vector and bit-manipulation operations near memory. PEI [4] proposes specialized fixed function blocks for graph workloads by extending the host ISA with specialized simple processing-in-memory operations such as atomic integer-increment and floating-point addition. While offloading such computations reduces application data movement through the memory hierarchy, having the host sequence multiple such operations requires a lot of communication and control to be transferred, incurring energy overheads. We demonstrate that both control and data locality are essential to improving energy efficiency.

MEALib has similar philosophical reasoning behind using accelerator libraries, although the authors aim to only accelerate fixed operations in Intel's MKL library [38]. Livia [52] proposes memory services and a task-based programming model, which lets programmers express the offloadable computations explicitly and the architecture dynamically places these tasks at various points in the memory hierarchy based on data locality. While new programming models are promising, they require the existing applications to be rewritten and not all workloads fit the task-based abstractions.

Control specialization: In LSSD [59], the authors propose specialization principles based on observations from domain-specific accelerators and present an accelerator design with a spatial fabric for exploiting concurrency and a low-power core for coordination for workloads with significant parallelism and defined coarse-grained work units. While they present an accelerator architecture, we propose an architecture and compiler framework with automated offload mechanisms that exploits analogous principles for computation cores across the memory hierarchy.

Charm [24] and camel [22] provide virtualization and composability of coarse-grain configurable accelerators with hardware resource management. RegionSeeker [81] targets application code

regions of varying granularities and identifies sub-graphs with control flow that are amenable to offload based on area constraints. These architectures do not specialize to reduce data or control movement. TOM [43] looks at transparently identifying code regions for GPU systems. While this is good for GPU workloads, it does not scale well for arbitrary workloads, as GPUs might prove to be power-inefficient if the resources are under-utilized. We propose an automated offload identification mechanism for arbitrary workloads with energy as primary metric and a common decentralized offload-model for memory-centric compute cores.

7 FUTURE WORK

This work proposes an architecture interface and an offload mechanism that enables independently coordinated fine-grained offload sequencing given the design space we have examined. This mechanism better matches future heterogeneous systems, where host overhead in spending time and bandwidth to get remote resources to perform an offload is higher. Traditionally, this overhead has been amortized by increasing the offload granularity. However, it does not opportunistically exploit all the specializable code regions since the control mechanisms do not match fine-grained vector offloads in cases where the computation engine is remote from the host.

While we have shown that a fine-grained memory-centric offload model that has both data and control locality can provide orders of magnitude better energy efficiency than an in-order core, there are extensions and directions of exploration for future inquiry which were not covered in this paper because of time constraints and tool limitations.

Extensions: The tradeoffs in specializing different types of control flows within an application can be studied to identify the hardware-software support needed to make these more amenable for better performance. Further, virtualization of compute engines in the memory hierarchy can help increase the degree of dynamism in where an offload will run, as also mentioned in Section 3.4.1. Other efficient accelerator architectures built on FPGA fabric can be explored further, given that the time-to-market of such programmable fabrics is shorter, and our framework enables compile-time generation of accelerator libraries with adaptive granularity that are suitable for being placed at different points in the memory hierarchy. The accelerator extraction mechanisms can further be augmented with the recent advances in HLS techniques to output core definitions with better pipelining and hardware library support [25, 42, 54, 55, 82]. We also intend to extend the evaluation platform to more deeply explore the use of both more aggressive (out-of-order) cores and heterogeneous general-purpose core types within the platform in combination with the offload accelerators. We plan to also explore the benefits of further compute specialization by identifying isomorphic patterns within the offloaded code regions of one or more application suites and synthesizing these as ASIC modules for increased energy efficiency and performance [19, 78].

Future Directions: This work presented a design space with compute and distance specializations as the main axes of differentiation among designs. This design space can be broadened with an additional dimension of *data specialization*, which aims to reorganize the data within memory [5, 36], across multiple memory elements [52] or with the help of dynamic data structures [51, 66] so as to reduce the adverse effects of the high memory access latency and data movement.

Given that future systems will have multiple loci of computation, the question then arises as to whether conventional cache hierarchies designed based on applications' reuse distance analyses are still the appropriate design point in a model that incorporates data specialization, localization, and custom compute engines, or whether an equally heterogeneous on-chip memory system will need to be developed to fully realize the potential benefits of these systems.

8 CONCLUSION

Energy efficiency in computing and data movement is becoming increasingly important. Both compute and distance specialization techniques must be co-designed for better energy efficiency. Driven by this, we make the case for a decentralized architecture framework that dynamically composes fine-grained accelerator definitions specializing both compute and control through the memory hierarchy to reduce data and control movement between various computation units. To achieve this, we first propose a generic architecture interface for supporting accelerators of flexible definitions and granularity with the ability to chain-invoke others. Our framework identifies acceleratable computations and control flows around the offload candidates from arbitrary applications. We assess the proposed decentralized offload decisions on multiple architectures with computation cores near cache and near memory. Across diverse classes of benchmarks, we see consistent benefits. Compared to an in-order core *with centralized datapath accelerator*, the energy efficiency is between 1.2×-3.9× and EDP improvement is between 1.37×-4.4× for applications with cache affinity, while for applications with low reuse distances and/or random memory accesses the energy efficiency is between 1.17×-2.55× and EDP improves by 1.4×-9.82×, thereby validating the promise of decentralization.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful suggestions. This work was funded in part by NSF award #1822923 (SPX: SOPHIA).

REFERENCES

[1] Richard Afoakwa, Lejie Lu, Hui Wu, and Michael Huang. 2019. To Stack or Not To Stack. In *International Conference on Parallel Architectures and Compilation Techniques*. IEEE, NY, USA, 110–123.

[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 481–492.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of the International Symposium on Computer Architecture*. ACM, NY, USA, 105–117.

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, NY, USA, 336–348.

[5] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-Stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, NY, USA, 131–143.

[6] Amogh Akshintala, Vance Miller, Donald E. Porter, and Christopher J. Rossbach. 2018. Talk to My Neighbors Transport: Decentralized Data Transfer and Scheduling Among Accelerators. *Proceedings of the 9th Workshop on Systems for Multi-core and Heterogeneous Architectures.* http://par.nsf.gov/biblio/10060951

[7] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 1–13.

[8] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. *International Journal of Parallel Programming*, 31, 6 (2003), 411–428.

[9] Erfan Azarkhish, Christoph Pfister, Davide Rossi, Igor Loi, and Luca Benini. 2016. Logic-base interconnect design for near memory computing in the smart memory cube. *IEEE Transactions on Very Large Scale Integration Systems*, 25, 1 (2016), 210–223.

[10] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2015. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, NY, USA, 1317–1322.

[11] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization*, 14, 2, Article Article 14 (June 2017), 25 pages.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. ACM, NY, USA, 72–81.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39, 2 (2011), 1–7.

[14] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, and Paolo Ienne. 2005. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *Design, Automation and Test in Europe*. IEEE, NY, USA, 1246–1251.

[15] Calin Cascaval, Siddhartha Chatterjee, Hubertus Franke, Kevin J Gildea, and Pratap Pattnaik. 2010. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54, 5 (2010), 5–1.

[16] Shuai Che, Arkaprava Basu, and Jonathan Gallmeier. 2016. Challenges of Programming a System with Heterogeneous Memories and Heterogeneous Processors: A Programmer's View. In *Proceedings of the Second International Symposium on Memory Systems*. ACM, NY, USA, 99–103.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE international symposium on workload characterization*. IEEE, NY, USA, 44–54.

[18] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, NY, USA, 33–38.

[19] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, NY, USA, 272–283.

[20] Nathan Clark, Amir Hormati, and Scott Mahlke. 2008. Veal: Virtualized execution accelerator for loops. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, NY, USA, 389–400.

[21] Jason Cong, Zhenman Fang, Michael Gill, Farnoosh Javadi, and Glenn Reinman. 2017. AIM: Accelerating Computational Genomics through Scalable and Noninvasive Accelerator-Interposed Memory. In *Proceedings of the International Symposium on Memory Systems*. ACM, NY, USA, 3–14.

[22] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. 2013. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *International Symposium on Low Power Electronics and Design*. IEEE, NY, USA, 305–310.

[23] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. Architecture Support for Accelerator-Rich CMPs. In *Design Automation Conference*. ACM, NY, USA, 843–849.

[24] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, NY, USA, 379–384.

[25] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Design Automation Conference*. IEEE, NY, USA, 1–6.

[26] George S Davidson, Jim R Cowie, Stephen C Helmreich, Ron A Zacharski, and Kevin W Boyack. 2006. *Data-centric computing with the netezza architecture*. Department of Energy, USA.

[27] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, NY, USA, 1221–1230.

[28] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-Memory Chip. In *Proceedings of the International Conference on Supercomputing*. ACM, NY, USA, 14–25.

[29] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, et al. 2016. Highly scalable near memory processing with migrating threads on the Emu system architecture. In *6th Workshop on Irregular Applications: Architecture and Algorithms*. IEEE, NY, USA, 2–9.

[30] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, NY, USA, 383–396.

[31] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 283–295.

[32] Jeremy Fowers, Kalin Ovtcharov, Michael K Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2019. Inside Project Brainwave's Cloud-Scale, Real-Time AI Processor. *Proceedings of the International Symposium on Microarchitecture*, 39, 3 (2019), 20–28.

[33] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 1–14.

[34] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 126–137.

[35] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The Terasys massively parallel PIM array. *Computer*, 28, 4 (1995), 23–31.

[36] Maya Gokhale, Scott Lloyd, and Chris Hajas. 2015. Near Memory Data Structure Rearrangement. In *Proceedings of the International Symposium on Memory Systems*. ACM, NY, USA, 283–290.

[37] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DYSER: Unifying functionality and parallelism specialization for energy-efficient computing. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 38–51.

[38] Qi Guo, Tze-Meng Low, Nikolaos Alachiotis, Berkin Akin, Larry Pileggi, James C Hoe, and Franz Franchetti. 2015. Enabling portable energy efficiency with memory accelerated library. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 750–761.

[39] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing. In *Proceedings of the International Symposium on Microarchitecture*. ACM, NY, USA, 12–23.

[40] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. In *ACM Transactions on Architecture and Code Optimization*. ACM, NY, USA, 1–25.

[41] Bruce Kester Holmer, David E Culler, and Alvin M Despain. 1993. *Automatic design of computer instruction sets*. Ph.D. Dissertation. Citeseer.

[42] Philipp Holzinger, Marc Reichenbach, and Dietmar Fey. 2018. A New Generic HLS Approach for Heterogeneous Computing: On the Feasibility of High-Level Synthesis in HSA-Compatible Systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. ACM, NY, USA, 18–27.

[43] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News*, 44, 3 (2016), 204–216.

[44] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture*. ACM, NY, USA, 1–12.

[45] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. 2012. Extending OpenMP*with Vector Constructs for Modern Multicore SIMD Architectures. In *OpenMP in a Heterogeneous World*. Springer, Berlin, Heidelberg, 59–72.

[46] Snehasish Kumar, Nick Sumner, Vijayalakshmi Srinivasan, Steve Margerm, and Arrvindh Shriraman. 2017. Needle: Leveraging program analysis to analyze and extract accelerators from whole programs. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 565–576.

[47] Monica S Lam. 1990. Instruction scheduling for superscalar architectures. *Annual Review of Computer Science*, 4, 1 (1990), 173–201.

[48] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*. IEEE, NY, USA, 75–86.

[49] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture*. ACM, NY, USA, 469–480.

[50] Jieun Lim and Hyesoon Kim. 2014. Design space exploration of memory model for heterogeneous computing. In *International Symposium on Computer Architecture and High Performance Computing*. IEEE, NY, USA, 160–167.

[51] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, NY, USA, 235–245.

[52] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, NY, USA, 417–433.

[53] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. 2016. Asic clouds: Specializing the datacenter. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, NY, USA, 178–190.

[54] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating parallel accelerators from parallel programs. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 245–257.

[55] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. 2015. Inter-procedural resource sharing in High Level Synthesis through function proxies. In *International Conference on Field Programmable Logic and Applications*. IEEE, NY, USA, 1–8.

[56] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *International symposium on high performance computer architecture*. IEEE, NY, USA, 457–468.

[57] Lifeng Nai and Hyesoon Kim. 2015. Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals. In *Proceedings of the International Symposium on Memory Systems*. ACM, NY, USA, 258–261.

[58] Siddharth Nilakantan, Karthik Sangaiah, Ankit More, Giordano Salvadory, Baris Taskin, and Mark Hempstead. 2015. Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, NY, USA, 278–287.

[59] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *International Symposium on High Performance Computer Architecture*. IEEE, NY, USA, 27–39.

[60] Nvidia. 2018. NVDLA Open Source Project. http://nvdla.org/index.html

[61] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberley Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. Intelligent RAM (IRAM): Chips that remember and compute. In *IEEE International Solids-State Circuits Conference. Digest of Technical Papers*. IEEE, NY, USA, 224–225.

[62] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *Hot chips*. IEEE, NY, USA, 1–24.

[63] Jerry L Potter. 1985. *The massively parallel processor*. Mit Press, USA.

[64] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. IEEE Press, NY, USA, 13–24.

[65] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization*. IEEE, NY, USA, 110–119.

[66] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting Dynamic Data Structures on Distributed-Memory Machines. In *ACM Transactions on Programming Languages and Systems*. ACM, NY, USA, 233–263.

[67] Robert Schilling, Thomas Unterluggauer, Stefan Mangard, Frank K Gürkaynak, Michael Muehlberghuber, and Luca Benini. 2018. High speed ASIC implementations of leakage-resilient cryptography. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, NY, USA, 1259–1264.

[68] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 67–80.

[69] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 273–287.

[70] Amirali Sharifian, Snehasish Kumar, Apala Guha, and Arrvindh Shriraman. 2016. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 1–14.

[71] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. 2016. Data-Centric Computing Frontiers: A Survey On Processing-In-Memory. In *Proceedings of the Second International Symposium on Memory Systems*. ACM, NY, USA, 295–308.

[72] Mukund Sivaraman and Shail Aditya. 2002. Cycle-Time Aware Architecture Synthesis of Custom Hardware Accelerators. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, NY, USA, 35–42.

[73] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration,* 58 (2017), 74–81.

[74] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. 2014. CortexSuite: A synthetic brain benchmark suite. In *IEEE International Symposium on Workload Characterization*. IEEE, NY, USA, 76–79.

[75] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. 2004. The MOLEN polymorphic processor. *IEEE transactions on computers,* 53, 11 (2004), 1363–1375.

[76] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *IEEE International Symposium on Workload Characterization*. IEEE, NY, USA, 55–64.

[77] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, NY, USA, 205–218.

[78] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, NY, USA, 163–174.

[79] Mingliang Wei, Marc Snir, Josep Torrellas, and R Brett Tremaine. 2005. *A near-memory processor for vector, streaming and bit manipulation workloads.* Technical Report. University of Illinois at Urbana-Champaign.

[80] Youngsik Kim, Tack-Don Han, Shin-Dug Kim, and Sung-Bong Yang. 1997. An effective memory-processor integrated architecture for computer vision. In *Proceedings of the International Conference on Parallel Processing*. IEEE, NY, USA, 266–269.

[81] Georgios Zacharopoulos, Lorenzo Ferretti, Emanuele Giaquinta, Giovanni Ansaloni, and Laura Pozzi. 2018. RegionSeeker: Automatically Identifying and Selecting Accelerators From Application Source Code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 38, 4 (2018), 741–754.

[82] Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, and Zhiru Zhang. 2016. Improving high-level synthesis with decoupled data structure optimization. In *Design Automation Conference*. IEEE, NY, USA, 1–6.